Computer Science 331

Basic Data Structures

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #9

Outline

Objectives for Today

Objectives for Today:

- Review of several basic data structures, including types of *arrays* and *linked lists*
- *Reference:* Text, Chapter 4

Assumption: You have seen most of this already! Some implementation and analysis details may be new.

Suggested Exercises for Later:

- Write specifications of requirements for the various operations being discussed
- Write a few of the algorithms sketched here in more detail
- Sketch proofs of correctness, and analyses of worst-case running times, using techniques from class

Static Array

A data structure providing access to a *fixed* number of data cells of some type

- Attribute — *length* : Number of data cells for which access is provided; this — and the type of data to be stored in cells — must be specified when the array is declared and cannot be changed
- Data cells have unique integer *indices* between 0 and $length - 1$
- The type of data that can be stored in each cell is called the *base type* of the array
- A data cell can be accessed *at unit cost* by specifying its index
- Many programming languages, including `Java`, directly support this data structure

Example

Suppose A is the following array of `String`'s:

0	1	2	3	4	5
a	c	x	g	h	`null`

- Length of A: 6
- Base Type of A: `String`
- Current value of $A[3]$: g
- Charge to access or store an entry of A at a given index: 1 unit

Automatic Initialization of an Array

An operation like

```
String[] sArray = new String[25];
```

declares the type of a variable (in this case, `sArray` — setting this to be an array that stores `String`'s) and sets the *length* of the array (in this case, 25)

Initial Value in Each Cell: The *default value* for the base type

- Default Value for Numeric Types: 0
- Default Value for `char` Type: $\backslash u$0000 (Unicode value of 0)
- Default Value for `boolean` Type: `false`
- Default Value for Class Types: `null`

Initialization of an Array with Values

Initial values can be enclosed in braces, separated by commas

- `A.length` automatically set to the number of initial values listed

Example: The statement

```
int[] age = { 2, 4, 7, 3, 6, 5 }
```

creates the following array

	0	1	2	3	4	5
`age:`	2	4	7	3	6	5

Cost To Initialize an Array: $\Theta(n)$, where $n = A.length$

- actual cost is some function $f(n) = an + b$ (a, b constants)
- $f(n) \in \Theta(n)$ (definition satisfied for $c_L = a$, $c_U = a + b$, and $N_0 = 1$)

Traversal of an Array

Visiting some or all of the cells in an array...

- Beginning at some index (usually 0)
- Going in either direction (usually by increasing index)

Since arrays allow direct access, implementing *traversals* is straightforward:

```
for (i=0; i<A.length; ++i) {
  // process array entry A[i]
}
```

Worst-Case Cost for a Traversal: $\Theta(nT(n))$, where $T(n)$ is the worst-case cost to process `A[i]`

Special Case: Finding a Given Value

Strategy:

- Traverse array from index 0
- Compare each array element with the given value until it is found or all entries have been checked
- Return index if the value is found; throw an exception or return an exceptional value (eg, -1) otherwise

Since at most a constant number of steps are used at each array index, the worst-case cost is: $\Theta(n)$

Replacing an Element of an Array (by position)

Replacing the Element at Position *i*

- Given an index i and value v, replace contents at position i with v

How To Do This: `A[i] = v`

Error Conditions: $i < 0$ or $i >= A.length$

Worst-Case Cost: $\Theta(1)$

- actual cost is a function $f(n) = c$ (c a constant)
- $c \in \Theta(1)$ (definition satisfied by $c_L = c$, $c_U = c$, and $N_0 = 1$)

Replacing an Element of an Array (by value)

Replacing One Value with Another

- Given values v and w, replace w with v in the array, or report that v was not found

How To Do This:

- Find index i such that $A[i] = w$ or report that w is not in the array. Cost: $\Theta(n)$
- Set $A[i] = v$. Cost: $\Theta(1)$

Error Conditions: none

Worst-Case Cost: $\Theta(n)$ (cost of the search function dominates)

- $f(n) = c_1 + (c_2 n + c_3) + c_4 \in \Theta(n)$

Additional Operations for Storage of Sets

Suppose now that an array is used to store a **set**:

- Elements of a set — and the values in the currently used part of the array — are distinct
- New attribute: *numElements* — size of the set currently stored
- *Requirements: numElements* $\leq$ *length* and the set's elements are stored at positions $0, 1, \ldots, numElements - 1$
- Default values for base type are stored at positions $numElements, numElements + 1, \ldots, length - 1$

Insertion of an Element into an Array

Operation: Given a value v, add v to the represented set

Error Conditions: $numElements = length$ (array is already full)

Situations of Interest:

- Storage order of elements in the array is unimportant *and* the new element is guaranteed *not* to be in the set already
- Storage order of elements in the array is unimportant *but* it is possible that the "new" element is already in the set
- Storage order of elements in the array is important

Insertion of an Element into an Array (Case 1)

If Storage Order is Unimportant and the New Element is Guaranteed Not To Be in the Set:

How To Do This:

- If $numElements = A.length$, report that A is full.
- Otherwise, set $A[numElements] = v$ and increment $numElements$.

Worst-Case Cost: $\Theta(1)$

Insertion of an Element into an Array (Case 2)

If Storage Order is Unimportant But the Element Might Be in the Set Already:

How To Do This:

- If $numElements = A.length$, report that A is full. Cost: $\Theta(1)$
- If there exists an index i such that $A[i] = v$, report that v is already in A. Cost: $\Theta(n)$
- Otherwise, set $A[numElements] = v$ and increment $numElements$. Cost: $\Theta(1)$

Worst-Case Cost: $\Theta(n)$ (cost of the search dominates)

Insertion of an Element into an Array (Case 3)

Insertion if Storage Order *is* Important:

How To Do This:

- If $numElements = A.length$, report that A is full.
- Otherwise, "shift" all elements from the insertion location "up" one position in the array and copy the new element into its correct spot.

Worst-Case Cost: $\Theta(n)$ (inserting into location 0)

Deletion of an Element from an Array

Operation: Given a value v, remove v from the represented set

Error Conditions: v is not in the array

Deletion if Storage Order is Unimportant:

- Find index i such that $A[i] = v$ or report that v is not in the array.
- Set $A[i] = A[numElements - 1]$; decrement $numElements - 1$.

Worst-Case Cost: $\Theta(n)$ ($\Theta(1)$ to delete, but $\Theta(n)$ to find v)

Deletion if Storage Order is Important

- Find index i such that $A[i] = v$ or report that v is not in the array.
- "Shift" all elements at index $i + 1$ to $numElements - 1$ one position "down"; decrement *numElements*.

Worst-Case Cost: $\Theta(n)$ (deleting element 0)

Dynamic Arrays

Lengths of *dynamic arrays* can be changed as needed

Java (and a few other languages) support dynamic arrays

Reasons To Use a Dynamic Array:

- it may be difficult to derive a rigorous upper bound on the number of elements that will be stored in the array,
- extra memory is not available (or expensive), so allocating a large static array with an excessive number of unused entries is not feasible.

Changing the Length of a Dynamic Array

To change the length of an array A to *newLength* (from the text):

1. Define an array *temp* with the same base type as A and with length *newLength*.
2. Use `System.arraycopy` to copy the contents of A into *temp*.
3. Set $A = temp$.

Warning: This is fine if the base type is an elementary type (eg, `int`, `char` or `boolean`). More work may be needed and, possibly, `System.arraycopy` should not be used, if the base type is a class — because it may not be obvious *how* the array elements should be copied over in this case!

For More Information: Search for "deep copying versus shallow copying" online.

Changing Array Length When Representing a Set

Very Bad Idea: Resize the array every time the set size changes

This is a bad idea because:

- too expensive — *each* operation costs $\Theta(n)$ due to the resizing

A Much Better Idea: Keep the array length linear in the set size.

- Eg. Contract the array by one half if fewer than one-third of array entries are used; double the array size when it fills up

This is better because:

- *amortized* cost is $\Theta(1)$

Why not contract if one-half of elements are used and double when full?

Linked Data Structures

Consist of zero or more *nodes* that are allocated as-needed and that are connected via references or pointers

- *Advantage:* Structures can grow as needed, unlike static arrays — and at low cost, unlike dynamic arrays
- *Disadvantage:* Constant-time direct access (by index or position) is not supported
- *Reference:* Sections 4.4-4.6 of the text includes an extensive discussion including Java implementations

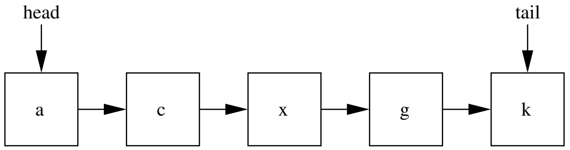
Singly Linked Lists

Brief Description: Nodes are Linearly Connected — each has a *value* and a reference to its *successor* node

Attributes:

- *head*: Reference to the first node in the list
- *tail*: Reference to the last node in the list (optional)
- *length*: Number of nodes in the list

Example:

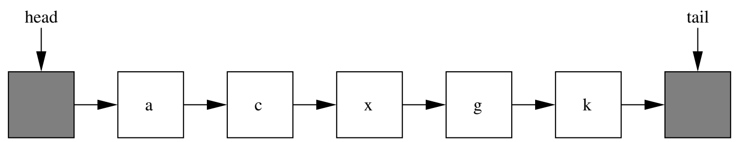


Singly Linked Lists with Dummy Nodes

Singly Linked Lists with Dummy Nodes:

- *Variation:* Nodes at head (and tail) do not store values — they are placeholders
- *Motivation:* Simplifies implementation of some operations

Example:



This variant, but without the tail node, is implemented in the textbook.

Initialization of a Linked List

How To Do This:

- Allocate a dummy node for the tail (both value and successor set to `null`).
- Allocate a dummy node for the head (value set to `null`, successor set to `tail`).
- Set length to be 0.

Worst-Case Cost: $\Theta(1)$

Traversal of a Linked List

How To Do This:

- Initialize a “cursor” to the head node’s successor.
- While the cursor is not equal to the tail of the list.
  - Visit the node pointed to by the cursor.
  - Set cursor to its successor.

Worst-Case Cost: $\Theta(n)$ (constant number of operations done per node)

Application: Finding a Given Element

Searching by Value:

- *How To Do This:*
  - Traverse the list from the beginning; halt once the value being searched for is found.
- *Worst-Case Cost:* $\Theta(n)$ (worst-case requires traversing the entire list)

Searching by Position:

- *How To Do This:*
  - Traverse the list from the beginning; halt once the desired position is reached.
- *Worst-Case Cost:* $\Theta(n)$ (worst-case is searching for the last element in the list)

Replacing an Element of a Singly Linked List

How To Do This:

- Traverse the list from the beginning; halt once the value to be replaced is found.
- Overwrite the value stored in the current node with the new value.

Worst-Case Cost: $\Theta(n)$ (cost of finding the element to be replaced dominates)

Insertion of an Element (Case 1)

If Storage Order is Unimportant and the New Element is Guaranteed Not To Be in Set:

How To Do This:

- Create a new node whose value is the element to insert, and whose successor is set to the successor of the head node.
- Set the head node’s successor to the new node.

Worst-Case Cost: $\Theta(1)$ (constant number of steps)

Insertion of an Element (Case 2)

If Storage Order is Unimportant But the Element Might Be in the Set Already:

How To Do This:

- Traverse the entire list to check whether the element is already in the list. Cost: $\Theta(n)$
- If the element is not in the list, insert it at the head. Cost: $\Theta(1)$

Worst-Case Cost: $\Theta(n)$ (dominated by the cost of the search)

Insertion of an Element (Case 3)

If Storage Order is Important:

How To Do This:

- Traverse the list from the beginning to find node (cursor) that should come *before* the new node.
- Set the new node's successor field to the successor field of the cursor.
- Set the cursor's successor field to the new node.

A Complication:

- If the new node goes at the beginning of the list, it is inserted after the (dummy) head node (no traversal required).

Worst-Case Cost: $\Theta(n)$ (inserting at the tail)

Deletion of an Element

How To Do This:

- Traverse the list from the beginning to locate the node to delete (target) *and* its predecessor.
- Set the predecessor's successor node to the target's successor node (thus "unlinking" the node pointed to by target from the list).
- Need the tail's predecessor in addition to the tail itself in this case.

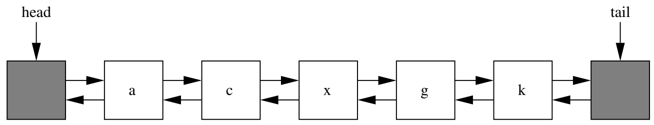
A Complication:

- Don't forget to set the target's value field to `null` (to make sure that the actual data is deleted)

Worst-Case Cost: $\Theta(n)$ (deleting the last element in the list)

Doubly Linked Lists

Variation: Nodes now have references to their *predecessors* as well as their *successors*



Advantage:

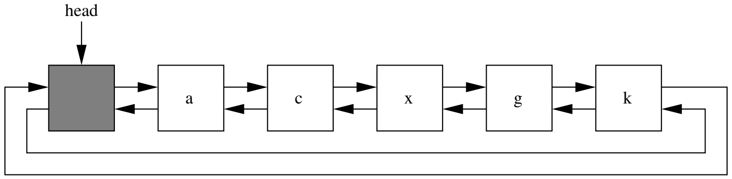
- Coding simplified (node's predecessor easily found)

Disadvantages:

- extra storage overhead for the additional predecessor references
- more difficult to code

Circular Lists

Variation over Doubly-Linked List: Replace pair of dummy nodes with a single one



Advantage over Doubly Linked List:

- slightly less extra storage (only one dummy node)